# Runtime Code Polymorphism as a Protection Against Side Channel Attacks

Damien Couroussé, Thierno Barry, Bruno Robisson, Philippe Jaillon, Olivier Potin, Jean-Louis Lanet

# Runtime Code Polymorphism as a Protection against Side Channel Attacks

Damien Couroussé[1], Thierno Barry[1], Bruno Robisson[2], Philippe Jaillon[3], Olivier Potin[3], and Jean-Louis Lanet[4]

[1] Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LIST, MINATEC Campus, F-38054 Grenoble, France
[2] CEA-Tech DPACA, Gardanne, France
[3] École Nationale Suprieure des Mines de Saint-Etienne, France
[4] INRIA de Rennes

**Abstract.** We present a generic framework for runtime code polymorphism, applicable to a broad range of computing platforms including embedded systems with low computing resources (e.g. microcontrollers with few kilo-bytes of memory). Code polymorphism is defined as the ability to change the observable behaviour of a software component without changing its functional properties. In this paper we present the implementation of code polymorphism with runtime code generation, which offers many code transformation possibilities: we describe the use of random register allocation, random instruction selection, instruction shuffling and insertion of noise instructions. We evaluate the effectiveness of our framework against correlation power analysis: as compared to an unprotected implementation of AES where the secret key could be recovered in less than 50 traces in average, in our protected implementation, we increased the number of traces necessary to achieve the same attack by more than 20000×. With regards to the state of the art, our implementation shows a moderate impact in terms of performance overhead.

## 1 Introduction

Side channel attacks are an effective means to recover a secret, by the observation of physical phenomena related to the secured activity. From the knowledge of the program under attack (e.g. the AES cipher), the attacker will try to establish a correlation between the observation traces and hypothesis about the intermediate values used during the secret computation (e.g. the output of the first SBOX computation). The hypothesis that provides the best correlation value is then used to recover the secret (e.g. the value of the AES key). Usually, a few points in the observation traces exhibit good correlation values with the hypothesis, which correspond to the *leakage point*, i.e. the time when the secret is observable during the computation.

Two main protection schemes are effective against side channel attacks: *hiding* and *masking*. The key idea of masking is to split the sensitive values of the secured computation in several shares, in order to break the correlation between

the observations and the hypothetical intermediate values. To recover the secret key from a masked implementation, and provided that the shares are computed at different times, an attacker needs correlation analysis of higher orders, i.e. analysis involving several observation points simultaneously. However, higher order attacks present a computational complexity that grows exponentially with the order of the attack; they are therefore more difficult to use in practice. Hiding consists in moving the point of information leakage both in time (during the secured activity) and in space (location of the activity on the chip), and in reducing the amplitude of the observable leakage. Indeed, side channel correlation analysis relies on precise spatial and temporal control of the target, and the effectiveness of the attack is strongly correlated to the amount of spatial and temporal variation in the observation signal [13]. Spreading the point of leakage over different times or places over many executions will drastically reduce the effectiveness of the attack, requiring more observation traces and/or more powerful analyses to recover the secret. In practice, robustness against side channel attacks is provided by a combination of hiding and masking countermeasures.

We define polymorphism as the capability to regularly change the behaviour of a secured component at runtime without altering its functional properties. By modifying the temporal and spatial properties of the observations of the attacked target, polymorphism increases the difficulty to perform the correlation analysis used in side channel attacks. Hence, it can be understood as a hiding countermeasure.

Non-deterministic processors [15] achieve what we call *polymorphic execution*, i.e. the shuffled execution of a program from a static binary input residing in program memory. May et al. achieve dynamic instruction shuffling [15] and random register renaming [14]. Bayrak et al. [6] describe an instruction shuffler: a dedicated IP inserted between the processor and the program memory (instruction cache). Nowadays, many Secure Elements integrate similar features in order to de-synchronise observation traces and to decrease the signal to noise ratio. Dedicated hardware designs offer a better security to performance ratio, at the expense of a higher cost. With the fast growing market of the Internet of Things, software-only solutions are appealing since they could be more easily adapted to the wide range of product architectures available, and are upgradeable.

By software means only, [1,10] propose to compile a program that contains several functionally equivalent execution paths, where one of the execution paths is randomly selected at runtime. This approach reduces the overhead on execution time at the expense of an increased program size. Amarilli et al. [3] were the first of our knowledge to exploit compilation of code variants against side channel attacks. The program is compiled before each execution thanks to a modified static compiler, in order to shuffle instructions and basic blocks at runtime. Their approach is shown to increase the number of traces of DPA on AES by a factor of 20. The work of Agosta et al. [2] is the closest of our work. They present a code morphing runtime framework that involves register renaming, instruction shuffling and the production of semantically equivalent code fragments.

In this paper, polymorphism is implemented with runtime code generation driven by random data. The key idea is to regularly generate new versions of the secured binary code on the target. Each program version is functionally equivalent but has a different implementation, so that each execution would lead to a different observation. The polymorphic code generator is produced by a framework for runtime code generation adapted to the constraints of embedded systems (section 2). We detail the mechanisms used to bring variability in the generated code by selecting random registers, randomly selecting semantically equivalent instructions, reordering instructions, and inserting noise instructions. We provide an experimental evaluation of the effectiveness of our approach against Correlation Power Analysis (CPA) on a software implementation of AES, and show that execution time and code size overheads are compatible with the memory and computation capabilities of constrained embedded targets (section 3). We present related works of our knowledge in section 4, and conclude in section 5.

## 2 Runtime code polymorphism for embedded systems

### 2.1 Overview of `deGoal`

`deGoal` is a framework for runtime code generation. Its initial motivation is the use of runtime code specialisation to improve program performance, e.g. execution time, energy consumption or memory footprint. In this section, we first sketch the characteristics of `deGoal` and then present how we extended it for the purpose of security.

In classical frameworks for runtime code generation such as interpreters and dynamic compilers, the aim is to provide a generic infrastructure for code generation, bounded by the syntactic and semantic definition of a programming language. The generality of such solutions comes at the expense of an important overhead in runtime code generation, both in terms of memory footprint and computing time and computing energy. In `deGoal`, a different approach is used: code segments (thereafter called *kernels*) are generated and tuned at runtime by ad hoc runtime code generators, called *compilettes*. Each compilette is specialised to produce the machine code of one kernel. Syntactic and semantic analyses are performed at the time of static compilation, and compilettes embed only the processing knowledge that is required for the runtime optimisations selected. As a consequence, compilettes offer very fast code generation (10 to 100 times faster than typical frameworks for runtime code interpretation or dynamic compilation), present a low memory footprint, can run on small microcontroller architectures such as 8/16-bit microcontrollers [5], and are portable [8].

The building and the execution of an application using `deGoal` consist in the following steps as illustrated in Figure 1: writing the source code using a mix of `C` source code and of our dedicated `cdg` language; compiling the binary code of the application and the binary code of compilettes; at runtime, generating the binary code of kernels by compilettes and in the end running the kernels.

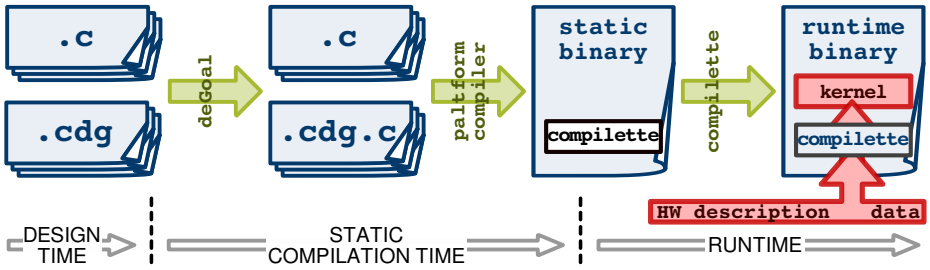


Fig. 1: `deGoal` workflow: from the writing of application's source code to the execution of a kernel generated at runtime

`Cdg` is an assembly-like DSL. From the programmer's perspective, it represents a major paradigm shift: `Cdg` expressions describe machine code that will be *generated* at runtime instead of instructions to be executed. Compilettes are implemented by using a mix of ANSI `C` and `Cdg` expressions. The `C` language is used to describe the control part of the compilette that will drive code generation, while `Cdg` expressions perform code generation. The `Cdg` instruction set includes a variable-length register set, extensible by the programmer. From this high-level instruction set, compilettes map the `Cdg` expressions to machine code according to (1) the characteristics of the data to process, (2) the characteristics of the execution context at the time of code generation, (3) the hardware capabilities of the target processor, (4) execution time and/or energy consumption performance criteria. In all cases, code generation is fast, produces efficient code, and is applicable to low-resource embedded systems such as micro-controllers [5].

### 2.2 A polymorphic SubBytes function

We introduce in this section the implementation of the SubBytes function of AES as a tutorial introduction to `deGoal`, before presenting in greater details the internals of runtime code generation. It is also the protected implementation used in our experiment (section 3).

The code generator is implemented with `Cdg` like any other code generator designed in `deGoal`. At this stage, polymorphic code generation is a feature provided by the code generation framework but has no need to be explicitly controlled by the programmer. In listing 1.1, the function `gen_subBytes` is a standard C function, its implementation being translated from `Cdg` to plain C source code before the static compilation.

`deGoal` allows to mix `Cdg` expressions for code generation, enclosed between the delimiters `#[` and `]#`, and C code. Moreover, C expressions, enclosed in `#( )`, can be inserted inside `Cdg` expressions; they will be evaluated at the time of code generation, i.e. at the time the compilette is executed. The program code of the `SubBytes` routine is written in a memory buffer at the address contained in the variable `code` (lines 1 and 5). Lines 6-7 declare the instantiation of typed variables that will be mapped at the time of runtime code generation to physical

Listing 1.1: Implementation of the compilette for the SubBytes function of AES

```
1  void gen_subBytes (cdg_insn_t* code,
2                     uint8_t* sbox_addr, uint8_t* state_addr)
3  {
4  #[
5    Begin code Prelude
6    Type uint32 int 32
7    Alloc uint32 state, sbox, i, x, y
8    mv state, #(state_addr)
9    mv sbox, #(sbox_addr)
10   mv i, #(0)
11   loop:
12     lb x, @(state+i)  //  x := state[i]
13     lb y, @(sbox+x)   //  y := sbox[x]
14     sb @(state+i), y  //  state[i] := y
15     add i, i, #(1)
16     bneq loop, i, #(16)
17   rtn
18   End
19 ]#;
20 }
```

registers. The variables `state` and `sbox` store respectively the address of the AES state and the address of the Sbox (C variables `state_addr` and `sbox_addr` line 2). A label (`loop`, line 11) defines the starting location the loop over the 16 state bytes. The variable `i` stores the loop index. It is initialised at line 10 and used as an offset value for the load and store instructions, respectively at lines 12 and 14, where the notation `@(a+k)` (lines 12 to 14) denotes an indirect memory access to the address stored in the variable `a`, offseted by an address variable `k`. The Sbox substitution is produced at line 14, where the temporary variable `y` is loaded with the memory contents at address `sbox` offseted by `x`. The `Cdg` instruction `rtn` (line 17) generates the termination code of the SubBytes routine, and the `Cdg` instruction `End` (line 18) terminates the code generation: it flushes the live instructions remaining in the instruction scheduler (section 2.3), and emits the machine code of backward branches for which the branch destination address cannot be calculated during the first code generation pass (not detailed in this paper).

### 2.3 Implementation of code polymorphism with deGoal

In this work, we re-target the original purpose of `deGoal` in order to focus on security aspects: we exploit the flexibility provided by `deGoal` for runtime code generation to achieve runtime code polymorphism. A program code produced by the polymorphic code generator is thereafter called *polymorphic instance*.

**Input**: the list of available registers $regs$
**Input**: the id of the physical register allocated $reg$
**Output**: regs
▷ `Get a random register among the list of available registers`
$l \leftarrow$`Length(`$regs$`)`
$i \leftarrow$ `Rand(`$0, l-1$`)`
$reg \leftarrow regs[i]$
▷ `Remove register` $i$ `from the list of free registers`
$regs \leftarrow$ `Delete(`$regs, i$`)`

**Algorithm 1:** Random Register Allocation

**Register allocation.** In dynamic compilers, instruction selection and instruction scheduling are usually performed before register allocation [12]. Despite allocation techniques used in dynamic compilers, such as linear scan [17], provide a reduced computational cost as compared to graph colouring usually used in static compilers, they are still out of reach of the computational power available in the platforms that we target. Hence, in a compilette, register allocation is done first, before instruction scheduling, by using a greedy algorithm (algorithm 1). Our purpose is to lighten the pressure on instruction selection and instruction scheduling: if register allocation is done first, it becomes possible to perform instruction scheduling from a much simpler intermediate representation: our allocator simply needs to maintain a list of the free registers available.

**Instruction selection.** Instruction selection is performed after register allocation for the motivations detailed previously. Instruction selection is done at the level of `Cdg` expressions: each expression can be mapped to one or more machine instructions depending on the target processor architecture and code generation options (e.g. favour code compactness or code execution time). Instruction selection is implemented with `switch ... case` segments driven by random values. For the purpose of achieving code polymorphism, we introduce supplementary variants to provide more opportunities for polymorphism. The semantic equivalences used in the context of the experiment presented in this paper, that are possibly selected by instruction selection are described below (`r` denotes a random value):

```
c := a xor b <=> c := ((a xor r) xor b) xor r
c := a xor b <=> c := (a or b) xor (a and b)
c := a - b   <=> k := 1 ; c:= (a + k) + (not b)
c := a - b   <=> c := ((a + r) - b) - r
```

We emphasize on the fact that, despite the number of semantic variants introduced for random instruction selection is low in the current experiment, adding new instruction variants will only have an impact on the memory footprint of the code generation library embedded onto the target but not on the execution time of code generation. In other words, adding more selection variants will grow the size of the `switch ... case` segments used in instruction selection, but does

**Input**: the instruction buffer $B$ (a circular buffer)
**Input**: the instruction to insert $I$
**Output**: $B$

```
▷ look for insertion slots, from the last buffer instruction
pos ← Tail(B)
while I has no data dependence with B[pos] do
  | pos ← Prev(B[pos])
end
▷ randomly insert the new instruction
i ← Rand(pos, Tail(B))
B ← insert I at position i in B
▷ flush the first instruction if B is full
if Prev(Head(B)) is equal to Tail(B) then
  | emit instruction at Head(B)
  | Head(B) ← Next(Head(B))
end
```

**Algorithm 2:** Instruction Shuffling

not impact the performance of branching to one of the cases from the `switch` expression.

**Instruction scheduling.** The process of instruction selection, presented above, produces instructions in a bounded ordered instruction buffer that behaves like a FIFO. At this stage, the instruction buffer contains the machine instruction encodings, and the description of *defs* and *uses* registers (i.e. modified and read registers, respectively). The instruction buffer is implemented with doubled-chained lists. This clearly has a strong impact on the manipulation cost of the data structure, but we have chosen this implementation to maximise the number of insertion opportunities versus execution efficiency. The experimental section shows that the performance of polymorphic code generation remains good.

In traditional compilers, instruction scheduling aims at improving the performance of code execution: machine instructions are ordered in program memory in order to minimise execution time, in particular the number of processor idle cycles. The difficulty of scheduling lies in finding a possibly optimal ordering of machine instructions without breaking the semantics of the original source program. To achieve this, resource constraints, control-dependence and data-dependence need to be considered.

In the case of code polymorphism, our aim is to exploit scheduling opportunities to generate many variants of the same source program, all functionally equivalent and semantically correct. Code performance is only a secondary matter with regards to these objectives. Hence, we consider resource constraints that impact code correctness, but not those that only impact program performance. Instruction scheduling is performed in one pass (algorithm 2): each time a new instruction is inserted in the instruction buffer, first the list of possible insertion slots is computed, by comparing the defs and uses of the inserted instruction

and of the instructions already stored in the buffer. Next, the insertion position is randomly selected among the list of insertion slots previously identified. If no insertion slot was found, the new instruction is appended at the end of the instruction buffer. If the instruction buffer is full, its first instruction is emitted in program memory to free one instruction slot.

Pipeline hazards [16], which only affect program performance but not program correctness, are not considered in our scheduling policy. Our main purpose is to lighten the computational cost of scheduling, but this choice has an other interesting effect: processor stalls, which are an observable effect of pipeline hazards on the execution of a program, are likely to add a supplementary source of temporal variation in the observation of program execution. Still in the aim of fast code generation, control-dependence constraints are simply handled by considering control instructions as scheduling barriers: when a control instruction is met, the whole instruction buffer is flushed.

**Insertion of noise instructions.** Noise instructions are appended to the end of the instruction buffer *before* the insertion of a new instruction (section 2.3). $n$ noise instructions are inserted with probability $p$, where $n$ is randomly selected in a configurable uniform discrete distribution [1; N]. If needed, extra registers are randomly selected among the free registers. If no register is available, registers are randomly selected, pushed and popped on the stack before and after use. Several kinds of instructions can be inserted: core arithmetic instructions that execute in one processor cycle (e.g. integer operations `add` or `sub`) or in several processor cycles (e.g. multiply and load `mla` on ARM Cortex-M cores) and memory accesses to a data table possibly provided by the user (e.g. the SBOX lookup table).

## 3 Experimental evaluation

### 3.1 Experimental setup

We used the STM32VLDISCOVERY evaluation kit from STMicroelectronics. The board is fitted with a Cortex-M3 core running at 24 MHz, provides only 128 kB of flash memory and 8 kB of RAM, and is not equipped with hardware security protections. All the binary programs are produced from the `arm-none-eabi` GNU/gcc toolchain in version 4.8.1 provided by Code Sourcery, using the compilation options `-O3 -static -mthumb -mcpu=cortex-m3`. Therefore, we compare our implementation with the fastest reference implementation that we could obtain from the target compiler.

The side-channel traces were obtained with a 2208A PicoScope, which features a 200 MHz bandwidth and a vertical resolution of 8 bits. We use an EM probe RF-B 3-2 from Langer, and a PA 303 preamplifier from Langer. The sampling acquisition is performed at 500 Msample/s over a window of 10000 samples.

Our reference implementation is an unprotected 8-bit implementation of AES that follows the NIST specification, and all the round functions are generated at

runtime, in RAM, by a polymorphic code generator specialised for this implementation of AES, as introduced in section 2. All the polymorphic mechanisms presented above are activated in the code generator. The probability of inserting noise instructions is set to $p = 1/8$, and the number of inserted instructions $n$ is selected in the uniform discrete distribution $[1; 8]$. A new polymorphic instance can be generated every $\omega = \{1, 10, 100, 1000, 10000\}$ executions of AES, but for the side channel attack we use the shortest code generation interval, $\omega = 1$.

### 3.2 Model of attack

We perform the attack on the output of the SubBytes function in the first AES round. On the contrary to most first order CPA analysis where the synchronisation point is put at the beginning of the AES encryption, we consider the case where the attacker is able to create a synchronisation point at the beginning of the SubBytes function. To ease the temporal alignment of the measurement traces, a trigger signal is generated via a GPIO pin on the board, held high during the execution of the SubBytes function. Using this setup, the security evaluation is performed with stricter conditions since the execution variability of the polymorphic AddRoundKey function does not contribute to the variations in the observation traces.

The code generator itself could be also the target of attacks even it does not access to the value of the secret key. Such attacks are however out of the scope of this paper and are left for future works.

### 3.3 Code generation interval

The code generation interval $\omega$, defined in equation 1, is related to the frequency at which a new polymorphic instance is generated as compared to the number of executions of the target kernel (in our experiment, the function SubBytes). $\omega$ is a value with no unit, comprised between 1 and $+\infty$. $\omega = 1$ represents the case where a new polymorphic instance is generated for each execution of the target kernel, and $\omega \to +\infty$ the case where a polymorphic instance is generated only once at startup, which is equivalent to the use of statically generated code. Our hypothesis is that, the closer $\omega$ is to 1, the more difficult a physical attack should be, because of the lesser probability that the observation of two executions will appear correlated. On the other hand, the overhead incurred by code generation is more important as $\omega$ is smaller.

$$\omega = \frac{\text{nb. executions}}{\text{nb. code generations}} \tag{1}$$

### 3.4 Correlation power analysis

We perform a first order CPA against both unprotected and protected implementations. We model the electromagnetic emission with the Hamming weight of one byte of the output of the first SubBytes function. The analysis computes

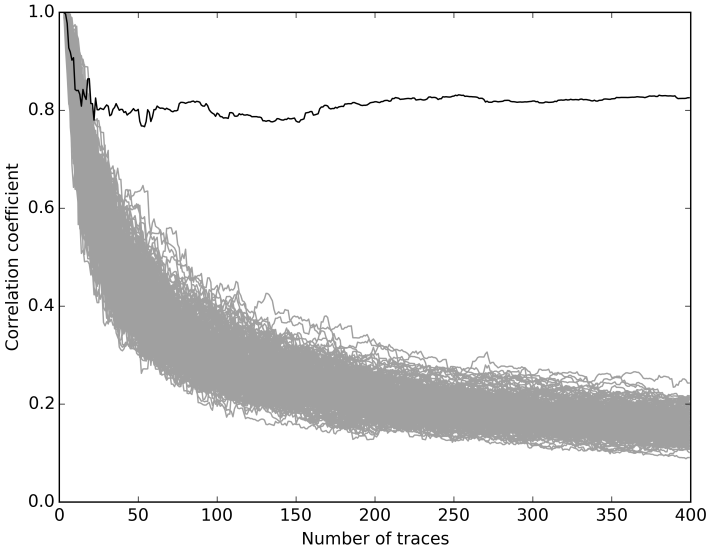


Fig. 2: Correlation values from the CPA attack on the unprotected AES

the sample estimation $r$ of Pearson's correlation coefficient $\rho$ between each trace and the model, for each possible hypothetical value of the involved key part.

Figure 2 presents the results of our CPA attack on the unprotected implementation. The correct key distinguishes from all the other hypothetical key values as soon as 35 traces with correlation values above 0.8. This result validates the experimental setup and the choice of the Hamming weight model used in the correlation analysis. As an illustration example of the impact of polymorphism, we show in figure 3 the results of a CPA attack on our polymorphic implementation, for a code generation interval of $\omega = 200$, i.e. far above the number of traces necessary to recover the AES key on the unprotected implementation. In this case, the first 200 traces are obtained from the execution of the same polymorphic instance of AES, which explains why the correlation value of the correct key clearly distinguishes from the other key hypotheses. After 200 executions, a new polymorphic instance is generated and executed for the next 200 executions. The impact of polymorphism on the correlation traces is clearly visible: the correlation of the correct key hypothesis suddenly decreases after 200 traces. After 300 traces, the correct key hypothesis is no longer distinguishable from the other key hypothesis because the correlation analysis merges the execution traces from two AES instances that behave differently. This also illustrates the fact that, in practice, the code generation interval $\omega$ must have a value strictly below the number of traces required to recover the key from an unprotected implementation. This setting is however strongly depending on the nature of the target and the practicability of the attack on an unprotected implementation.

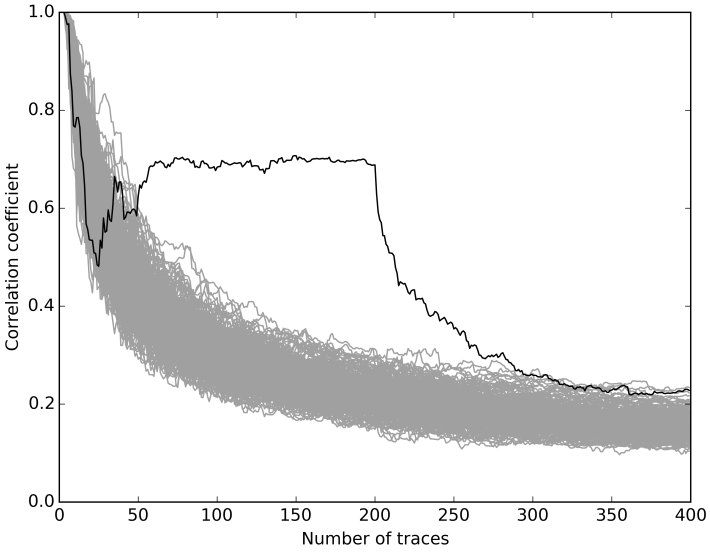


Fig. 3: Correlation values from the CPA attack on the polymorphic AES, $\omega = 200$

When a new polymorphic instance is generated after each execution ($\omega = 1$), 100000 traces are necessary to recover the key with a success rate of 50% (Figure 4). A success rate of 100% is reached after 120000 traces.

### 3.5 Execution time overhead

To estimate the cost added by our protection to a reference implementation, we measure the execution time overhead $k$ (Equation 2), where $t_{\text{ref}}$, $t_{\text{gen}}$ and $t_{\text{poly}}$ respectively denote the average execution time of the unprotected reference implementation, the average execution time of the polymorphic runtime code generation and the average execution time of the polymorphic instance. Our measure of the execution time overhead takes into account the increase of the execution time due to the execution of the polymorphic instance (which has a suboptimal code as compared to the reference unprotected implementation) and due to runtime code generation. In this measurement, we consider that the overhead incurred by code generation is distributed over $\omega$ runs.

$$k = \frac{t_{\text{gen}} + \omega \times t_{\text{poly}}}{\omega \times t_{\text{ref}}} \tag{2}$$

Table 1 compares the execution times of the unprotected AES and several variants of the polymorphic AES. In this section, we name the *full polymorphic AES* our implementation where all the round functions are protected with polymorphism. The unprotected version executes in 5320 processor cycles. We observe the execution time of the polymorphic versions over 1024 runs. The full

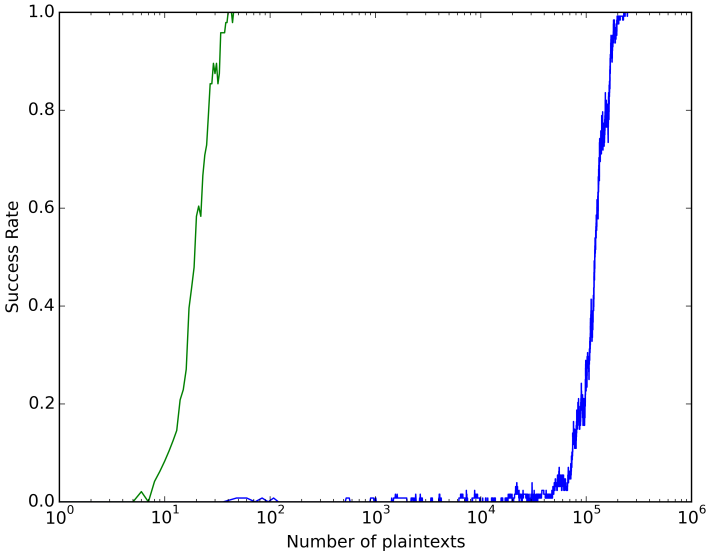


Fig. 4: Success rate of the CPA attack on the unprotected (green, leftmost) and the full polymorphic implementation (blue, rightmost), $\omega = 1$.

polymorphic AES executes in 10487 to 13696 processor cycles (average 12211 cycles). Table 1 also illustrates the fact that, if only one round function is protected with polymorphism (AddRoundKey or SubBytes), the execution time overhead is reduced. The execution time of the unprotected version is perfectly stable over measurements because of the relative simplicity of the micro-architecture of our experiment target. However, in contrast, the execution time of the polymorphic versions presents important variations. Considering the stability of the execution time of the unprotected version, this variability can only be accounted to implementation variations in each polymorphic instance.

Table 2 presents the overheads $k$ computed for the execution time measurements detailed in table 1, for different values of the code generation interval $\omega$. For a full polymorphic AES, in the case a new polymorphic instance is generated before *each* execution of AES ($\omega = 1$), we measure an average overhead of 20.10 (worst case 26.16). However, the overhead quickly decreases when the code generation interval is increased, because the cost of runtime code generation is distributed over several executions of the same polymorphic instance. Considering the number of traces required to recover the key from the unprotected AES, a code generation interval of $\omega = 10$ could be an exploitable version. In this case, the execution time overhead is only of 4.36 in average (worst case 4.85).

Table 2 also illustrates the cost incurred by the polymorphic instances *only*. For large code generation intervals (e.g. $\omega = 10000$), the contribution of runtime code generation becomes negligible in the overall overhead, i.e. the overhead only represents the cost of executing the polymorphic instances as compared to

Table 1: Execution times (in cycles, measured for 1024 executions of each configuration) of the AES function ($t_{\text{exe}}$) and of the code generator ($t_{\text{gen}}$) for the unprotected version (Unprotected), the AES with a polymorphic AddRoundKey function only (AddRoundKey), the AES with a polymorphic SubBytes function only (SubBytes), and all four round functions polymorphic (All round functions).

| | Unprotected | | | AddRoundKey | | | SubBytes | | | All round functions | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | min. | avg. | max. | min. | avg. | max. | min. | avg. | max. | min. | avg. | max. |
| $t_{\text{exe}}$ | 5320 | 5320 | 5320 | 5565 | 5977 | 6269 | 5850 | 6090 | 6340 | 10487 | 12211 | 13696 |
| $t_{\text{gen}}$ | 0 | 0 | 0 | 10894 | 20166 | 27970 | 24922 | 32604 | 41497 | 95265 | 109842 | 127269 |

Table 2: Execution time overhead for AES in the same conditions as table 1. The reference is the unprotected version, which runs in 5320 cycles.

| | AddRoundKey | | | SubBytes | | | All round functions | | |
|---|---|---|---|---|---|---|---|---|---|
| | min. | avg. | max. | min. | avg. | max. | min. | avg. | max. |
| $\omega = 1$ | 3.16 | 4.91 | 6.37 | 5.81 | 7.27 | 8.94 | 20.10 | 22.94 | 26.16 |
| $\omega = 10$ | 1.32 | 1.50 | 1.66 | 1.59 | 1.76 | 1.92 | 3.86 | 4.36 | 4.85 |
| $\omega = 100$ | 1.09 | 1.16 | 1.22 | 1.16 | 1.21 | 1.25 | 2.17 | 2.50 | 2.78 |
| $\omega = 1000$ | 1.09 | 1.13 | 1.18 | 1.16 | 1.15 | 1.20 | 2.17 | 2.32 | 2.59 |
| $\omega = 10000$ | 1.05 | 1.12 | 1.18 | 1.11 | 1.15 | 1.19 | 1.99 | 2.30 | 2.58 |

the reference unprotected implementation. Indeed, the code of the polymorphic instances is less optimal, in terms of execution time, because of the mis-ordering of instructions, the use of suboptimal (and longer) code sequences and the execution of useless instructions.

### 3.6 Memory footprint

Table 3 reports the memory footprint of our work, including the size of the code generators and the memory size reserved for code generation. The full polymorphic implementation presents an extra overhead of 16 KB of program size, but it is possible to reduce the memory footprint by partially applying polymorphism to AES: if only AddRoundKey and SubBytes are polymorphic, the memory overhead is of 8 KB only.

## 4 Related works

The work of Agosta et al. [2] is the closest of our work: they were the first to gather in a same runtime code transformation framework the use of semantically equivalent code sequences, random register allocation, instruction shuffling and array permutations as a protection against side channel attacks. The code transformations are applied to the 64 `xor` instructions of a 32-bit implementation of AES based on OpenSSL. Due to the higher number of traces required to extract the key from an unprotected implementation (11600), the exploitable

Table 3: Memory footprint (in bytes) of complete programs running the unprotected AES and the polymorphic AES in the same variants as presented in table 1

| | text | data | bss | total |
|---|---|---|---|---|
| Unprotected | 6144 | 40 | 772 | 6956 |
| AddRoundKey only | 8128 | 56 | 2948 | 11132 |
| SubBytes only | 7540 | 56 | 2948 | 10544 |
| AddRoundKey + SubBytes | 10964 | 88 | 3980 | 15032 |
| Full polymorphic AES | 16984 | 88 | 6028 | 23100 |

code generation intervals are higher (between 100 and 3000) than in our work. The execution times of the polymorphic code instance and of the code generator are respectively 1.07× and 392.5× the execution time of the unprotected AES, and they report an overhead of 5× for $\omega = 100$. In our work, all the instructions all the AES program are protected with polymorphism, where the overhead is only of 2.50× in average for $\omega = 100$; in the worst case $\omega = 1$, code generation is 23.9× the execution time of the reference unprotected AES.

Other works present the design of polymorphic programs without intervention of runtime code generation, by random selection of functionally equivalent execution paths. Boulet et al. [7] describe the protection of Java Cards against side channel reverse engineering. The method applies to interpreters in virtual machines; it describes the random association and selection of functionally equivalent codes in the interpreter of the virtual machine. As compared to our work, this work does not use random register allocation, and instruction shuffling between code sections. Similarly, Crane et al. [10] propose to randomly switch the execution between different copies of program fragments to protect programs against cache side-channel attacks. The fragment variants are generated offline by a modified compiler, involving the insertion of nop instructions and random memory accesses. During program execution, the fragments are dynamically selected by trampolines with table-based random indirect branches. Their implementation targets a general-purpose multi-core desktop computer. Agosta et al. use a modified LLVM toolchain compiler to target ARM Cortex-M4 based microcontrollers [1]. Each instruction of the secured program section is replaced with several functionally equivalent instruction sequences. In addition, a masking scheme is applied for memory accesses and register spills. They report a performance overhead of 11× for the same cipher AES-S than in our study, however with code size overhead of 9×.

The execution of noise instructions (also called dummy instructions) is another known technique to introduce random time delays in order to mitigate side-channel attacks. In software, the execution of dummy instructions is achieved either by branching to a dedicated routine from predefined locations in the program (e.g. before, during and after the part of the code to protect) or by the triggering of a random delay interrupt. The routine for example executes a loop that decrements to zero the value of a randomly initialised register [9]. Hidden

Markov models [11] or pattern matching [18] are effective techniques to remove in the observation traces the parts corresponding to the execution or dummy instructions, or to create synchronisation points in the traces to cancel the misalignment created by random delays. However, these analyses rely on the fact that the inserted temporal noise presents a distinctive signature (for example the header of the interrupt handler). Ambrose et al. [4] propose to randomly insert a limited set of predefined instructions, similar to regular code instructions, that could also use a randomly selected register. They argue that such instructions, which modify the internal state of the processor, are of the same nature than the rest of the program instructions, hence causing higher power variations due to bit flips in registers. One of the contributions of our work is to extend this idea one step further: the noise instructions inserted in the program are of the same nature than real program instructions (arithmetic operations, memory operations, etc), and can target one or several free registers randomly selected. Furthermore, thanks to runtime code generation, we are able to better weave the noise instructions with the rest of the program as compared to the state-of-the art technique for dummy instructions.

## 5 Conclusion

We have presented a framework that achieves runtime code polymorphism as a generic protection against side channel attacks. Our implementation relies on a lightweight runtime code generation framework suitable for embedded systems, which usually out of reach of runtime code generation tools such as Just-In-Time compilers because of their low computing and memory resources. To the best of our knowledge, it is the first time that polymorphic runtime code generation could be achieved with such limited computing resources (8 kB of RAM and 128 kB of flash memory), with acceptable runtime overheads. Nevertheless, our implementation is applicable to cryptosystems and also to other software components that require some protections against physical attacks (e.g. pincode management, bootloaders, etc.), on a large range of computing platforms [8].

On our experimentation platform, we observed that the key can be recovered in less than 50 traces on an unprotected AES. However, on many real-life platforms, a side channel attack may require more traces to successfully extract a cipher key, even on an unprotected implementation: often than 10000 or 100000 traces. This means that, in practice, polymorphism can be effectively used with greater code generation intervals, so that the overhead of our approach becomes more tractable. Other design parameters, such as the parameters that control the insertion of noise instructions, will have an important impact both on the security margin and on the execution time and code size overheads.

Finally, we emphasise on the fact that polymorphism is not an end *per se*, but instead that it should be combined with other state of the art protections (e.g. masking). Provided the genericness of our approach and the lightness of our implementation, we consider that its integration in a more global security scheme is practical.

## Acknowledgments

This work was partially funded by the French National Research Agency (ANR) as part of the program Digital Engineering and Security (INS-2013), under grant agreement ANR-13-INSE-0006-01.

## References

1. Agosta, G., Barenghi, A., Pelosi, G., Scandale, M.: The MEET approach: Securing cryptographic embedded software against side channel attacks. IEEE TCAD 34(8), 1320–1333 (2015)
2. Agosta, G., Barenghi, A., Pelosi, G.: A code morphing methodology to automate power analysis countermeasures. In: DAC. pp. 77–82. ACM (2012)
3. Amarilli, A., Müller, S., Naccache, D., Page, D., Rauzy, P., Tunstall, M.: Can Code Polymorphism Limit Information Leakage? In: WISTP. pp. 1 – 21 (2011)
4. Ambrose, J., Ragel, R., Parameswaran, S.: Rijid: Random code injection to mask power analysis based side channel attacks. In: DAC. pp. 489–492 (2007)
5. Aracil, C., Couroussé, D.: Software acceleration of floating-point multiplication using runtime code generation. In: ICEAC. pp. 18–23 (2013)
6. Bayrak, A.G., Velickovic, N., Ienne, P., Burleson, W.: An architecture-independent instruction shuffler to protect against side-channel attacks. TACO 8(4), 1–19 (2012)
7. Boulet, F., Barthe, M., Le, T.H.: Protection of applets against hidden-channel analysis (2013), WO/2012/085482
8. Charles, H.P., Couroussé, D., Lomller, V., Endo, F., Gauguey, R.: deGoal a tool to embed dynamic code generators into applications. In: Compiler Construction, LNCS, vol. 8409, pp. 107–112. Springer (2014)
9. Coron, J.S., Kizhvatov, I.: Analysis and improvement of the random delay countermeasure of ches 2009. In: CHES. pp. 95–109. Springer (2010)
10. Crane, S., Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Thwarting cache side-channel attacks through dynamic software diversity. In: Network And Distributed System Security Symposium, NDSS. vol. 15 (2015)
11. Durvaux, F., Renauld, M., Standaert, F.X., van Oldeneel tot Oldenzeel, L., Veyrat-Charvillon, N.: Efficient removal of random delays from embedded software implementations using hidden markov models. In: CARDIS. pp. 123–140 (2013)
12. Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., Cox, D.: Design of the Java Hotspot client compiler for Java 6. TACO 5(1), 7:1–7:32 (2008)
13. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks: Revealing the secrets of smart cards. Springer (2007)
14. May, D., Muller, H., Smart, N.: Random register renaming to foil DPA. In: CHES, vol. LNCS 2162, pp. 28–38. Springer (2001)
15. May, D., Muller, H.L., Smart, N.P.: Non-deterministic processors. In: ACISP'01. pp. 115–129. Springer (2001)
16. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 4th edn. (2011)
17. Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Trans. Program. Lang. Syst. 21(5), 895–913 (1999)
18. Strobel, D., Paar, C.: An Efficient Method for Eliminating Random Delays in Power Traces of Embedded Software. In: ICISC, vol. 7259, pp. 48–60. Springer (2012)